A Report on the Assessment of a Software Engineering Process

Edward Bergman #15335633

Introduction

The role of a software engineer is to design software that fulfills the needs of a client. Software Engineers will follow the software engineering process to accomplish this task. This client, whether it be a financial institution, a volunteer group or even yourself, will have evolving needs in an ever evolving ecosystem. Fulfilling these needs is no longer the case of writing some software and shipping a finished product. Now more than ever, there is a need for well designed, well tested and well implemented software to ensure the product can adapt to these new concerns that may arise. There is an increasing need for continuous support on software, as technology advances and businesses grow, software must grow to meet their new needs. Building a software product that does not continuously run is fading . The use of microservices, the expanding ecosphere of open source and the increase in accessibility for anyone to produce software means a Software Engineer must design not only for the quality of their own software and their teams skills, but also that of others. With code in production that can be expected to run for countless years and relied upon by countless others, you can no longer call it a day and move on. Software must last, be reliable and expandable. This report will focus on how to assess a Software Engineering process and its ability to meet those requirements. The quantifiable data we can use, the tools available to process that data, the means by which this data is evaluated and finally the concerns of using this data to assess this process and a person's or team’s ability.

The Data

For this report we will focus on a few specifics of the Software Engineering process, namely code quality as judged by testing, planning and goals and finally technical debt. It is important to recognize there is other data we will not discuss, some we can measure and some we can not, or at least not in a simple manner.

We will cover this topic with regards to these questions:

1. What is the data?
2. How is this data generated / gotten?
3. How does this data correlate to the assessment of a software engineering process?

Testing

Testing is a well known aspect that correlates with quality Software. Testing is the means by which we can reiterate on code and be sure we have not broken anything. It allows us to identify where the software requires fixing or improvement and is overall a good source of data to assess the quality of the software engineering process in practice.

Testing	Obtaining	Inferring
Test Coverage	Tools can measure the amount of code covered by tests. Test types include, unit tests, integration tests, path tests and more.	The more coverage there is, the more we can modify the code safely.This implies a better process, that allows for high coverage.
Test Success	Most testing frameworks will give you measure of how many tests passed.	Test success implies the software is built correctly which is clearly an important factor of a software product.
Reproducibility	We can obtain the reproducibility of tests by running them several times, ensuring the same input will receive the same output.	Usually reproducibility scores are an indicator of the quality of async code in a system, the lower the ratio, the higher the quality of code.

Technical Debt

This data refers to the amount of work needed to fix what was a short term solution initially and will require more work in the future to create a more viable long term solution. An example of this could be an initial solution that is highly specific to a certain input but a more flexible solution that can handle more input would be desirable and perhaps needed in the future. This is technical debt, work will be required in the future to remedy this initial solution. Technical Debt, while largely considered a negative, it is necessary to ensure deadlines can be met and initial proof-of-concepts can be done, a desirable trait in a software engineering process. A good process will introduce enough time to plan and re assess those plans as time goes on. It is important to consider the impacts on technical debt introduced by the process used.

Technical Debt	Obtaining	Inferring
Duplication	Analyse a code base to identify code that can be considered duplicated.	Duplication can be considered a bad practice as it is a contributor to bugs in long-term projects. Less duplication can mean better software.
Complexity	Complexity is another subjective issue. A common method is Cyclomatic Complexity but other methods do exist.	Complexity can make software difficult to reason about. Less complex software is indicative of well structured code planned by a good process.
Dependencies	Dependencies introduce external points of potential failure. We can analyze the codebase to see the dependencies.	While dependencies are a necessary part of software, minimizing multiple dependencies for a single unit infers again a well planned software product.
Test Coverage	Covered in our testing section, test coverage can be obtained by analyzing the codebase.	Making sure test coverage remains high ensures a process can continue to run smoothly and minimize technical debt.
Documentation	Documentation can be analyzed in the codebase by having specific documentation rules in place.	Code will be read more than it is written. Ensuring good documentation minimized debt needed to re-understand code from scratch.

Planning, Communication and Quality Control

A good process will ensure that that workload is distributed in well defined small units of work which is done by planning. This ensures a engineers can be highly focused on specific problems/features. A process must also facilitate efficient and focused conversation. Whether this be meetings on specific units of work or for project direction, efficient communication can help identify problems quickly and address them and reduce the time wasted with back and forths. Lastly, this work must be of a reasonable quality and a process must have some way of trying to ensure this. Low quality work will introduce technical debt which we discussed previously and contribute to a bad culture of future low quality work. These 3 concepts are highly subjective and vary from project to project, but they can all be placed on a scale i.e. 0 planning is bad, over planning *can* be bad, therefore there is some scale to quantify a good level of planning. A good process will set these scales for themselves and try to ensure they all meet an “ideal” amount.

P, C & QC	Obtaining	Inferring
Planning	Most software engineering process will include some form a project manager. The time they spend planning can be used as data. Also, for retrospective grading of a process, we can compare the units of the plan where the goals were met and how many failed.	A good process will provide ample time for planning to occur. It will provide time for replanning to happen once the project develops further and modifications to the plan may need to be made. The more goals that are met vs not met, the more productive the planning is, hence the better the process.
Communication	Data for communication is a tricky one to quantify. For some projects, this may not be as important as it is for others. We can gather information on time dedicated to meetings, progress reports, emails/messages etc..	0 communication is bad and there exists an ideal range of communication. This varies on a project to project basis. An ideal process will minimize the amount of communication needed but still provide ample time for where it is needed.
Quality Control	We can produce some data on the code’s quality by having code reviews and automated build systems that ensure some form of code quality that is judged to be needed.	The higher quality of code produced, the better the process is for facilitating this high-grade code. Providing facilities and including methods for controlling code quality in the process is a sign of a good process.

The Tools

There are many tools available to the industry to assess the Software Engineering process. We will focus on the tools that take into account the data we have specified to provide some feedback on how to improve the process or where bottlenecks may exist.

Most if not all of the data listed in the previous section can be gotten manually, given that the required information is tracked (codebase, meetings, code reviews etc..). What tools give us are the ability to store this information and also process it to give us the data we need to infer what we need. A lot of this also revolves around automation of tedious tasks, such as testing and analyzing. Below are a list of tools which help us with these tasks and can provide insight into the software engineering process.

TestingWhiz

https://www.testing-whiz.com/

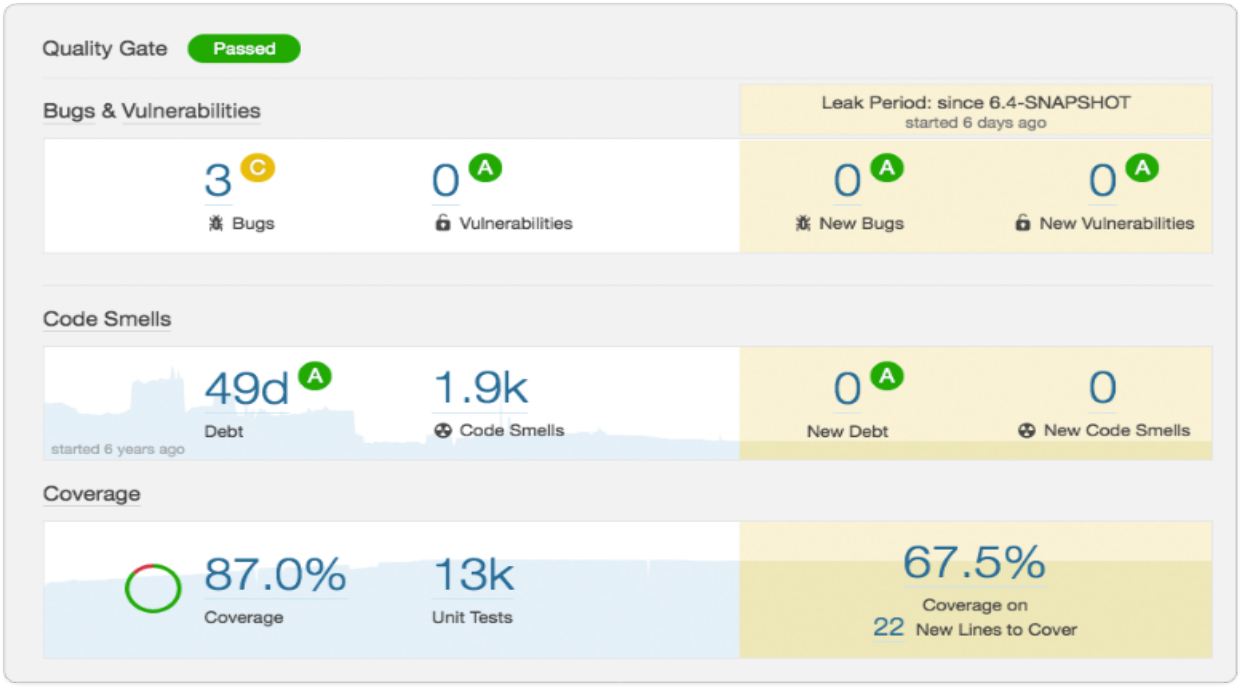
This is an automated testing tool for various different type of software projects, including web, mobile and server based projects. You can set up test suites of various kinds, for every layer of a system. This allows you to get a whole and comprehensive view of the system and its testing statistics. It provides many different ways to automate tests but more importantly, it also provides feedback on how those tests performed and the various sets of data we mentioned in our Data section on testing. It also provides many other tools to gather other statistics we have not mentioned such as info on bugs and tracking test efficiency. We can use all of this data to assess how our software engineering process has impacted our code quality. A solid process will use the automated testing as way to continuously track the quality of the code. If there is not enough data, the process will allow for more work to be dedicated to testing the code, meanwhile if there is enough data, we can assess the relative quality of our code base and adjust the plan if and as needed.

SonarQube

https://www.sonarqube.org/

SonarQube is a tool that applies large amount of static analysis to codebases to provide insights into many aspects of the code. One of the major features of this open source software is its ability to highlight problems and calculate the technical debt of a project. This software can also analyze build files for better code understanding, as well as be integrated with any continuous integration systems you may have in place. This software is flexible enough to both function as a management tool but also as part of the continuous integration process itself. Users can get in depth insights into where the technical debt is coming from and the estimated work hours required to fix it. This tool also provides other tools for monitoring code quality such as searching for bugs, test coverage and duplication.

Along with all these figures, it also provides a lot of data visualisations to help understand all these figures in a relative manner. For a tool that assesses the code quality of a project, it can provide very deep insights into the software engineering process in use and how it affects code quality. It allows a project manager to understand the relations between the process in use and how it directly impacts code quality.



Toggle

https://toggl.com/

Toggle is a time management tool, aimed at understanding how your time is spent. While this could apply to many different scenarios, they target their features and functionality towards employees and managers who wish to understand how much time they spend on certain things and the effects of that time spent. It provides many seamless integrations, allowing for a manager to understand their teams time as well. This tool can help you plan on how to spend your time productively and monitor the actual time spent on a task. Overall it can provide meaningful insights into the planning and communications aspect of a software engineering project.

The Algorithms

Defining what is a good and what is a bad software engineering process is a tricky subject. Even harder is to place its effectiveness on a scale as we saw with all the discussed data points. Designing an algorithm that will assess a process is even more difficult. We are faced with two major problems.

1. What works for one business/team may not work for another.
2. How do you quantize the software engineering process.

Let us start with our second point, explaining theoretical ways this can be done. We will then continue on and describe methods for tailoring the process efficiency as needed for different team. Lastly, we will investigate different ways we can obtain an actual breakdown of the process in such a way we can learn how to improve it. While this section does not cover discrete algorithms for assessment, it is important to realise that again, different business' will require different processes that can't be compared on a 1:1 basis. A company focused on small software with short deadlines require a high degree of productivity but technical debt may be of secondary concern. In contrast, companies such as Google or Tesla will require software with a high degree of quality and minimal debt. They are concerned with modifiable software that will run continuously where any downtime could cost them millions.

Putting a number on it

You can put a number on different aspects of the software engineering process but at the end of the day, these are all components of the whole picture. These metrics are useful for identifying where bottlenecks may exist and improving those areas but they do not give us the full picture. To fully evaluate a software engineering process in use, we must consider all these factors and decide what is important. A company desiring longevity in their software would wish to minimize technical debt while mission critical software requires code that is guaranteed to perform as expected, emphasizing the need for a well tested system. In an ideal world, we would judge all processes the same and attempt to maximize all positive scores and minimize all negative scores. We do however not live an ideal world so tradeoffs will take place, these tradeoffs should be modelled into our "Score" for a software engineering process, again trying to minimize the the overall negative outcomes and increase positive ones. All of these tradeoffs are a result of time. With infinite time, we would hopefully produce the ideal software solution. That is all very well and good but how do you go about turning all this information into a numerical score.

Without too much detailing on the specifics, let us generalize how you might go about coming up with your own algorithm. The score is gotten as a result of all our data points we can get, including ones we have not mentioned. Applying some function to all this data can give us a score.

Algorithm(TestingScore, TechnicalDebt, ManagementScore, X) = "Score"

This is a very abstract definition, where X is other data we have not mentioned and the other three inputs are simply a function of specific categories of data points we can get. In a very simplistic model, we could have a linear combination model and simply add all these scores together to get a total score for our process. Simple right? This sadly is very inaccurate as it would leave out a lot of detail. For example, our technical debt is very much related our testing score as well as other inputs. It also leave little room for emphasis on particular needs in a software engineering process, namely the need for different aspect scores to play a bigger role in our total score. This abstract function does not do the whole

problem justice as there could arguably by hundreds of inputs that change daily. It also does not factor in the requirements that could change in a company as time goes on. Assuming you had a function that could give you the most accurate representation of your score for your process, what happens when your company now needs to produce software that requires a high degree of security? This is something your business never had to worry about. Now your emphasis should shift more towards how well tested your code is and include new metrics which you may never have dealt with before. We could rethink our abstract function and perhaps add a greater bias towards these new parameters but long term, this is not a solution we can rely on nor can we consider it accurate enough to change how we might do things.

We can see now how difficult of a problem this can be, but thankfully there are solutions to these problems nowadays.

The Use of Intelligence

When a business is faced with problems on how to structure the processes they have in place to maximise their goals, they would employ a business analyst or have an external consultant who will drive change in the right direction (hopefully). There is an increasing use of a rapidly developing branch of computer science, Artificial Intelligence. Whether you can consider the actual implementation as intelligence or not, that is up to the reader. The end result is we can use this artificial intelligence to analyze problems and query for information we need. For a business, this might be a case of understanding how to maximise profits or increasing product use. They can employ what is known as Business Intelligence, feeding in any relevant data and getting a result indicating where focus should be diverted to maximise these scores. Evaluating any Software Engineering Process is no different, we can provide all our data and have some ‘intelligence’ recommend where to put more resources or where changes could be made to increase our score, whether it be for productivity, reliability, maintainability or some focus on all three. To understand how we might use Computational Intelligence to improve our process, we must first have some intuitive understanding of how this intelligence is able to analyze our process much better than we can ourselves.

There are multiple branches of artificial intelligence which ease strive to solve different categories of problems. For our case, we will use what is known as neural networks. There is nothing inherently intelligent about neural networks but they do something which could require a lot of intelligence, intuition and work to perform manually. A neural network is essentially some software with the purpose of recognizing patterns with as little pre-knowledge as needed. For a basic understanding, we can think of a neural network as our abstract function previously provided. You give it a bunch of inputs and it can provide you with some score. There is a few things we must think about though.

1. How does the network learn from our data?
2. How does this score actually compare to a “real” score?
3. How can this network adapt to our changing needs?

Let us address the first question as this is part of our understanding of a neural net. For a quick answer, this neural network will change the “abstract function” that we mentioned in an attempt to generate a score for which we can use to assess our process and begin to improve it. As for how this is done, essentially the network can be thought of as attempting to curve fit between all these data points as accurate as possible. This will provide us with a function that can for any set of inputs, tell us its relative score compared to other inputs. To put this into real terms, does increasing planning time result in a higher productivity score or should we increase the amount of time spent on reducing technical debt to increase productivity score. Unfortunately this method does require some guidance, this neural net has no notion of productivity and requires previous data to learn how data correlates. Fortunately though, this is just more data, such as LoC, project times, money spent etc… In fact, this is a benefit for us as we can decide what productivity we consider important, we could have competing objectives of which we need to maximise both of them. This is where a neural net is in its prime. Using output data (our productivity scores) and our input data (our various metrics), we can produce a function that accurately represents the results of our software engineering process. This also answers our third question, if we indicate to our network that we now wish to maximize a new objective such as safety, it can improve its function to emphasize things such as planning and testing, producing higher quality code.

This still leaves us with a glaring question of what is this score and how does it compare to the real world. If you were to get a score of 95 vs a score of 100, this gives you no insight into your process. To truly get a sense of how a software engineering process is performing and improving, a network must obtain as much data as possible. The more real world datasets you can feed to a network, the more accurate a score the network can give you. If we were to obtain all the data we required from Google's software engineering process and considered this to be our ideal, we would train a network on google's data set and compare our relative score. Of course, the score of an ideal process is unknown, we must consider our score as a relative score which can only be compared to data we know about. This does not take away from the fact we can train a network on our process and learn how to improve it.

We will end our discussion here on the use of artificial intelligence to improve our software engineering process but there are further concerns which should be taken into account before using computational intelligence to refine and asses our software engineering process. To name but a few:

- Overfitting. Sometimes you reach a local maximum of potential improvement. The function the network will assess by knows best how to deal with a certain kind of model and anything outside this model will produce a potentially low score, even if productivity actually increases. The best ways to combat this is to provide more diverse datasets and to adjust the neural network relative complexity.
- Limited Data. If you are a new business, you do not have access to the data required to have a network inform you on where to improve. While neural networks are getting better at learning from fewer sets of data, we can overcome this by adopting an already well known process with publicly available data (e.g. Agile Development) and adjust this process as necessary to fit our own needs.
- Unique data. Suppose we are a business attempting to maximize profits while keeping code quality high. We could hire a consultant for a large sum to teach our engineers but how do we factor this into our data? How to quantify this relatively unique data and hope our network does not react poorly to it. This is where it is important to remember this should be used as advice and not canon. We will explore this in our ethics section.

The Ethics

Whenever we mix technology and people, we must always consider the effects of these interactions. Automatic assessment based on quantifiable data has many benefits but if this assessment is not handled with caution, there are many issues that may arise. We will explore a few scenarios to to identify the ethics concern of using some automatic assessment software to assess our software engineering process .

Our first and most clear example of where ethical concerns may arise is in essence, attempting to produce the highest score possible. A good algorithm, AI or otherwise, will help you identify where to improve your process. Sometimes this could mean investing in a better continuous integration system to detect bugs early which is a good improvement to make. Where the issue lies is when your bottleneck revolves around the work an employee does. Of course if we have our team work extra hours, we will be more productive. Dedicating an employee to only perform testing to improve our testing bottleneck could also improve our productivity. This is quantifiable data, the amount of work an employee does, that correlates to an improvement in our production. What we can't quantify is the long term effects on our employees. Relying on the algorithm for improvement means all factors that are not data points become irrelevant if not handled carefully. Our employees, working extra hours or working on something they might find mundane or boring could be detrimental to the employees well-being and happiness. In the short term, this may lead to a score boost but in the long term you are damaging the quality of work that will be produced. For a business concerned with a short turnover for software, who are not concerned with the long term can use this to exploit its short term employees or contractors that are hired.

Our second example is of the score maximising software engineer. This engineer wishes to prove their worth to the company and figures how to maximise the score part associated with themselves in the algorithm. This engineer is now the most valuable member of the team. The hundreds of lines deleted and re-added, the commits of every word added and the bugs introduced intentionally and fixed are all clearly very valuable to the business right? This is obviously not the case but it shows us possible ways an engineer might try to maximise their score through "cheating" the system. It does not enforce good practice and

if anything it introduces bad patterns into software development. This is up to the discretion of the engineer and managers to notice this behaviour but it opens up the opportunity for someone with a questionable set of ethics to unfairly gain from the use of some algorithm to assess the quality of the software engineering process in use.

Lastly, we introduce a new problem and that is for engineers with little experience. There are multiple studies that claim the productivity of software engineer's can vary by 10-fold if not more. For a company who might use such similar algorithms to assess an engineers capability, they might miss the gem. A newer engineer may not have as much knowledge in a specific area as one with 5 years experience in that field but has the capability to learn that same amount of knowledge in 1 year. Even if the engineer himself is not highly productive, they might increase moral in the team through positivity or very knowledgeable in theory but not practice, allowing other team members to benefit from this knowledge. It doesn't need to be pointed out this would be a huge gain for any business but solely using algorithms for this kind of assessment have the issue that they may overlook these hard to quantify features. The ethical issue that is relevant here is placing everyone upon the same scale using only the data that can be gotten, this scale does not account for many hard to measure traits.

To finish our discussion, we will mention a simple solution to these ethical concerns. All of these ethical issues raised stem from the use of the algorithm as the main source of truth. This opens up the door for those who would wish to exploit this or for others to be mistreated by the algorithm. At the end of the day, these algorithms are not perfect or in the case of AI, not even known (blackbox principle). Therefore, to alleviate this issue, we must take this algorithm as an advisor. Treat this algorithm as an additional source of information on making decisions and not as the single source of truth. Be aware of the algorithms that you use in your assessment, where its strengths are and where its weaknesses may lie. Technology is advancing far faster than we can reason about its effects and therefore we must treat it with this in mind. Realising that we can't let an algorithm take over our decision making process, merely a guide, is the key to using automatic assessment effectively.

Final Remarks

This report focused primarily on data we can gather and measure. For us to assess the software engineering process, we must be able to put concepts into a numerical format by which an assessment tool can process this data. Combining all the relevant quantifiable data, we can produce a detailed picture of how work is done in our process and where any bottlenecks and issues may lie. Providing an algorithm that can do this for us is not a simple task, there is a lot of data to consider and the priorities for this data may differ from business to business. We explored the use of modern AI, namely neural networks, to address this concern. Using neural networks to tailor an algorithm to our specific needs allows us to investigate the whole process in an informed way. Lastly, we explored the ethical concerns that may arise from using an automated process to assess our software engineering process. This algorithm provides much more insight into the process than a person could hope to do manually, but we must remember, it does not know all. This key lack of knowledge in unquantifiable data is an algorithms downfall. Recognizing these key weaknesses can help us develop a system by which we ourselves, can assess our software engineering process in the most informed manner possible. Armed with this knowledge, we can continue forward, expanding and improving the software engineering process as we go.